



Centrum voor Wiskunde en Informatica

**REPORT***RAPPORT*

A framework for query optimization to support data mining

R. Choenni and A.P.J.M. Siebes

Computer Science/Department of Algorithmics and Architecture

**CS-R9637 1996**

Report CS-R9637  
ISSN 0169-118X

CWI  
P.O. Box 94079  
1090 GB Amsterdam  
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.

SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum  
P.O. Box 94079, 1090 GB Amsterdam (NL)  
Kruislaan 413, 1098 SJ Amsterdam (NL)  
Telephone +31 20 592 9333  
Telefax +31 20 592 4199

# A Framework for Query Optimization to Support Data Mining

Sunil (R.) Choenni

Arno P.J.M. Siebes

CWI

*P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*

## Abstract

In order to extract knowledge from databases, data mining algorithms heavily query the databases. Inefficient processing of these queries will inevitably have its impact on the performance of these algorithms, making them less valuable. In this paper, we describe an optimization framework for an efficient processing of queries generated by different data mining algorithms. In this framework, we show how to take advantage of the physical organization of the database, the operators and the control structures used in an algorithm. Finally, we discuss how our framework fits into conventional query optimization frameworks.

*CR Subject Classification(1991):* Database systems (H.2.4) *query processing*, Information search and retrieval (H.3.3) *search strategies*.

*Keywords and Phrases:* data mining systems, search strategies, query optimization, physical database design.

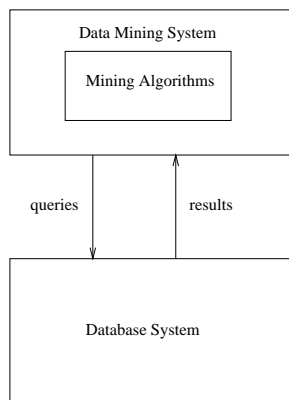


Figure 0.1: Integration of data mining and database systems

## 1. INTRODUCTION

The amount and kind of data that are stored in databases are still growing. These data may contain knowledge that can improve the quality of decisions taken in the present or future. Due to the huge amount and various kind of data that are stored in databases, conventional data analysing tools are inadequate to extract knowledge from these databases.

A field that deals with extracting knowledge from databases, without putting restrictions on the amount or types of data in a database, is data mining. Nowadays, data mining is achieving more and more interest from academy as well as from industry [FPSU 96]. The interest from academy stems from the many challenging research topics entailed by data mining. Interest from industry is due to the fact that data mining may result into knowledge that may be of vital importance for a company.

Research and development in data mining evolves in several directions, such as association rules, time series, and classification. The direction of association rules is focussed on the development of algorithms to find frequently occurring patterns in a database, see among others [AgIS 93, AgSr 94a, HoSw 95, MaTV 94, SrAg 96]. In time series databases, one tries to find all common patterns embedded in a database of sequences of events [AgSr 94b]. The classification of tuples in a number of groups on the basis of common characteristics and the derivation of rules from a group is another direction in data mining [AGII 92, HaCC 92, HoKe 94, AuVK 95].

Our interest lies in the last field. We are developing a system, within the scope of an Esprit project, to classify tuples in groups and to derive rules from these groups [WWVS 96]. The architecture of our system is according to Figure 1. A data mining system, equipped with several algorithms, generates queries and passes them to the database system. The database system derives the answers of the queries and passes them to data mining system, see Figure 1. In this architecture, the data mining system has as task to come up with strategies that limit the number of queries passed to the database system, and the database system has as task to process received queries. The advantages of such an architecture in the field of association rules have been discussed in [HoSw 95] and in the field of classification in [WWVS 96]. However, although search strategies attempt to minimize the number of queries passed to a database system in order to extract knowledge, they still generate a large number

of queries. Consequently, the architecture of Figure 1 is only viable if the data mining system receives the answers of queries within an acceptable amount of time.

This paper is devoted to the optimization of queries generated by different search strategies, which may appear in a data mining system. We show that queries generated at later steps in a search process are dependent of queries generated at earlier steps. As in [HaCC 92, HoKe 94], queries are regarded as conjunctions of predicates over a number of attributes. We assume that the number of tuples satisfying to queries are required for the search process and not the tuples self. We note that for many data mining algorithms this assumption holds, see among others [AuVK 95, AgIS 93, AgSr 94a, SrAg 96, HoKe 94]. We exploit these properties and present a framework for query optimization that seamlessly fits in traditional frameworks. For the time being, we have elaborated the framework for variants of three search strategies namely, hill climber, simulated annealing, and genetic algorithms.

The optimization techniques in our framework exploit the following aspects: physical organization of the database, the operators used in an algorithm, and the control structures of an algorithm.

We note that although some of the above-mentioned aspects have been exploited for query processing in several data mining algorithms, they neither have been wrapped up in a general optimization framework, nor their roles have been systematically analysed in the context of data mining algorithms. The optimization framework can be used by each data mining algorithm that satisfies to above-mentioned properties. What distinguish our framework from conventional frameworks for query optimization is that conventional ones do not benefit from the dependency between queries. Conventional frameworks optimize a query in isolation of other queries [Ullm 89, SWKH 76].

The remainder of this paper is organized as follows. In Section 2, we outline some preliminaries. In Section 3, we present a number of search strategies and study what kind of queries are generated by them, in order to discover knowledge. Then, in Section 4, we discuss how to optimize the queries generated by these strategies by storing and re-using intermediate results. Section 5 shows how the optimizing techniques of the previous section can be incorporated in existing frameworks for query optimization. Finally, Section 6 concludes the paper.

## 2. PRELIMINARIES & ASSUMPTIONS

In the following, a database consists of a universal relation. The relation is defined over some attributes, such as,  $att_1, att_2, \dots, att_n$ , and is a subset of the Cartesian product  $dom(att_1) \times dom(att_2) \times \dots \times dom(att_n)$ , in which  $dom(att_j)$  is the set of values that can be assumed by attribute  $att_j$ . A tuple is an ordered list of attribute values to which a unique identifier (tid) is associated. We assume that the content of the database remains the same during the mining process.

An expression is used to derive a relation and is defined as a conjunction of predicates over some attributes. The length of an expression is the number of attributes involved in the expression. Expressions with length 1 are called *elementary* expressions. An example of an expression of length 2 is  $< age \in [19, 24] \wedge gender = 'male' >$ , representing the males who are older than 18 and younger than 25. An expression  $e_{sub}$  is a subexpression of  $e$ , if each

elementary expression of  $e_{sub}$  is contained in  $e$  and  $length(e) > length(e_{sub})$ .

We deal with search spaces that contain expressions. An expression  $e'$  is an extension of  $e$  if  $e$  is a subexpression of  $e'$  and  $length(e') - length(e) = 1$ . An expression  $e'$  is called a reduction of  $e$  if  $e'$  is a subexpression of  $e$  and  $length(e) - length(e') = 1$ . An expression  $e'$  is called a *neighbour* of  $e$  if  $e'$  is an extension of  $e$  or  $e'$  is a reduction of  $e$ .

A *generalization* enlarges the range of an elementary expression, while a *specialization* reduces the range of an elementary expression. Examples of a generalization and specialization of  $\langle \text{age} \in [19, 24] \rangle$  are  $\langle \text{age} \in [19, 30] \rangle$  and  $\langle \text{age} \in [19, 20] \rangle$ , respectively.

Finally, we assume that the WHERE clause of a query consists of an expression, and that the output of a query is the number of tuples satisfying this expression. We note that this type of queries is quite significant for many data mining algorithms, see among others [AuVK 95, AGII 92, AgIS 93, AgSr 94a, SrAg 96, HaCC 92, HoKe 94, HoSw 95].

### 3. SEARCH STRATEGIES

To be successful, search strategies impose a certain structure on a search space [NaSS 86]. For example, a search strategy that is focussed on finding a local optimum in a search space that almost consists of local optima will not be very useful.

Unfortunately, the search spaces that stem from data mining problems neither have a specific structure nor the structure is known in advance. On the basis of evidence, one should choose for a search strategy. Therefore, a mining tool should be equipped with several search strategies. In this section, we discuss variants of a number of search strategies and we study the expected generation pattern of queries by each strategy. The search strategies are equipped with one or more operators that can be applied on expressions in the corresponding algorithm. We start the discussion with a variant of a hill climber, and continue with a variant of simulated annealing and genetic algorithm.

**Hill Climber** The variant of the hill climber discussed in this section is equipped with the operator *extension*, which takes as input an expression and computes an extension of it. The hill climber starts with an initial expression  $e_0$ . Then, it computes all extensions of  $e_0$  and their qualities, and the extension with the best quality becomes the expression for further exploration. The whole procedure will be repeated again until no improvements are possible, or some user defined criteria are met. We note that the expression with the best quality is the one with the maximal value. The structure of the procedure is given in Figure 2.

Searching according to a hill climber guarantees a local optimum. A local optimum is useful in the case that it does not deviate too much from a global optimum.

This variant of the hill climber is used in [HoKe 94]. Another variant of a hill climber with a generalization operator is used in [HaCC 92].

**Simulated annealing** The variant of simulated annealing discussed, in this section, is equipped with the operator *neighbour*, which takes as input an expression and computes an neighbour of it. In contrary to the hill climber, this strategy can choose with a certain probability an element for further exploration that has a worse quality than the current ele-

<pre> <b>procedure</b> Hill Climber <math>e := e_{loc} := e_0</math>; {initial solution} <math>q(e) := \text{quality}(e_{loc}) := \text{quality}(e)</math>; <b>repeat</b>   <b>repeat</b>     <math>e' := \text{extension}(e)</math>;     <math>q(e') := \text{quality}(e')</math>;     <b>if</b> <math>q(e') &gt; q(e_{loc})</math> <b>then</b>       <math>e_{loc} = e'</math>; <math>q(e_{loc}) := q(e')</math>;   <b>until</b> "extension is impossible"   <math>e := e_{loc}</math>; <b>until</b> "terminating criteria are met" <b>end</b>; </pre>	<pre> <b>procedure</b> Simulated annealing <math>e := e_0</math>; <math>q(e) := \text{quality}(e)</math>; <math>T := T_0</math>; {initial amount of time} <math>i := i_0</math> {initial number of iterations} <b>repeat</b>   <b>repeat</b>     <math>e' := \text{neighbour}(e)</math>;     <math>q(e') := \text{quality}(e')</math>;     <b>if</b> <math>q(e') &gt; q(e)</math> <b>or</b> <math>\text{rand}[0,1] &lt; \exp \frac{q(e) - q(e')}{T}</math> <b>then</b>       <math>e := e'</math>; {rand selects a number between 0 and 1}     <b>until</b> "loop has been repeated <math>i</math> times"     <math>T := \alpha * T</math>; <math>i = \beta * i</math>; {<math>0 &lt; \alpha &lt; 1</math> and <math>\beta \geq 1</math>}   <b>until</b> <math>T = 0</math> <b>end</b>; </pre>
--	--

Figure 0.2: (a) Hill Climber and (b) Simulated annealing

ment. This provides the possibility to escape from a local optimum. As time progresses, this probability gradually decreases until it becomes zero, which is the terminating criteria. In Figure 2(b), the structure of the procedure is outlined.  $\square$

**Genetic Algorithm** A genetic algorithm [Mich92] selects an initial population. Individuals in the population are represented as strings of bits. Then, it computes the quality of all individuals. On the basis of the quality a selection of individuals are made (an individual may be chosen more than once). Some of the selected individuals undergo a minor modification, called mutation. For some pairs of selected individuals a random point is selected, and the substrings behind this random point are exchanged, called cross-over. The selected individuals whether or not modified form a new generation.

A genetic algorithm has been described in [AuVK 95] to learn first order logic rules. An individual is represented as a sequence of predicates. The mutation operator has as effect that a predicate is generalized or specialized. Consider the following individual:  $\text{Pyramid}(X) \wedge \text{Color}(X, \text{yellow})$ . The mutation operator may change the color yellow into bright.

The crossover operation is implemented straightforward. Consider the following two individuals:  $\text{Pyramid}(X) \wedge \text{Color}(X, \text{yellow}) \wedge \text{Support}(V, W)$  and  $\text{Pyramid}(Y) \wedge \text{Color}(Y, \text{yellow}) \wedge \text{Support}(c, d)$ . A cross-over at the second position results into the individuals:  $\text{Pyramid}(X) \wedge \text{Color}(X, \text{yellow}) \wedge \text{Support}(c, d)$  and  $\text{Pyramid}(Y) \wedge \text{Color}(Y, \text{yellow}) \wedge \text{Support}(V, W)$ , respectively.  $\square$

From the above described strategies, we observe that expressions that will be evaluated in a next step depends on the present step, and *not* on former steps. This is a well-known property of a Markov process [GrSt 82]. This indicates that stored results of the present step may be used for the next step. For example, the tuples that satisfy an extension of an expression  $e$  will be always a subset of  $e$ . Searching for tuples that satisfy an extension of  $e$  in the set of tuples satisfying  $e$  will be, in general, cheaper than searching for those tuples in the database, since the database will contain more tuples. What intermediate results to store and how to reuse them is the topic of the next section.

#### 4. OPTIMIZATION

Query optimization is often performed in two phases, a so-called logical and physical optimization phase [Ullm 89]. In the logical optimization phase, it is determined in which order the involved (basic) operations in a query should be processed. In the second phase, it is determined how the basic operations can be efficiently performed. This depends on the way the data is stored, which is described in a physical schema [Choe 95]. If an inefficient physical schema is chosen for a database, this has its impact on query optimization. In Section 4.1, we discuss an efficient way to store a relation for data mining.

In the two-phased optimization process, it is assumed that queries are independent of each other, i.e., no profit is taken from the arrival pattern of queries. As a consequence, no reuse of information is made. We study how reusability can be exploited to support query optimization. To what extent, we may benefit from reusing intermediate results for query optimization depends on the operators used in a search strategy and the algorithm. In Section 4.1, we discuss the role of the operators, and in Section 4.2, we discuss the role of the algorithms.

##### 4.1 Physical schema

A relation will be stored as a binary storage model. In a binary storage model, there exists a separate table for each attribute, and each row in a table is a pair (attribute value, tid-list). The advantage of storing a relation as a number of binary tables is that queries requiring the number of tuples satisfying to an expression can be efficiently processed. To determine the number of tuples satisfying an elementary expression,  $att = 'v'$ , we access the binary relation corresponding to  $att$  with entry ' $v$ ' and count the number of tids in the tid-list. To determine the number of tuples satisfying a non-elementary expression, in which  $m$  attributes are involved, we access each of the  $m$  corresponding binary relations with the relevant entry, and save the tid-lists. Then, we intersect these tid lists, and count the resulted tids. In this way, activities as searching and retrieving of tuples are avoided.

In commercial database management systems, the binary storage model can be simulated by allocating an index to each attribute and sorting it on attribute value. An index can be regarded as a table, in which each row is a pair (attribute value, tid list).

##### 4.2 Operators

We discuss what information should be stored in order to optimize the basic operators, extension, reduction, generalization, specialization, and cross-over. We note that a neighbour can be regarded as either an extension or reduction, and a mutation as either a generalization or specialization. In the following, a list  $L_i^j$  contains the tuples identifiers (tids) satisfying the expression  $e_i^j$ .

**extension** Let us consider an extension,  $e$ , of expression  $\langle e_1^j \wedge e_2^j \wedge e_3^j \wedge \dots \wedge e_{n-1}^j \rangle$  with an expression  $\langle e_n^j \rangle$ . By keeping track of  $L_{1,2,3,\dots,n-1}$ , in which  $L_{1,2,3,\dots,n-1} = L_1^j \cap L_2^j \cap L_3^j \cap \dots \cap L_{n-1}^j$ , the number of tuples satisfying to the extension  $e$  can be computed by  $L_{1,2,3,\dots,n} = L_{1,2,3,\dots,n-1} \cap L_n^j$ , and counting the elements in  $L_{1,2,3,\dots,n}$ . We note that this observation was also made in [HoKe 94].  $\square$



**reduction** In contrary to the extension operator, reuse of intermediate results is not straightforward in case of the reduction operator. Consider the reduction  $e' = \langle e_1 \wedge e_2 \wedge e_3 \wedge \dots \wedge e_{j-1} \wedge e_{j+1} \wedge \dots \wedge e_n \rangle$  of the expression  $e = \langle e_1 \wedge e_2 \wedge e_3 \wedge \dots \wedge e_j \wedge \dots \wedge e_n \rangle$ . Then, the list of tids satisfying  $e$  can not be used in computing  $e'$ , since the tuples satisfying  $e'$  is not longer a subset of the tuples satisfying  $e$ . However, if tids of tuples satisfying proper subexpression are stored, e.g.,  $\langle e_1 \wedge e_2 \wedge e_3 \wedge \dots \wedge e_{j-1} \rangle$ , they can be used in computing the tuples satisfying a reduction. Subexpressions with length  $n - 1$  have the highest priority to be stored, since a reduction reduces an expression with length one.  $\square$

**Generalization and Specialization** Let us consider an expression  $e_i^j = att_i \in [v_k, v_p]$ . To determine the number of tuples satisfying to a generalization of  $e_i^j$ , we apply the following procedure. We determine the values that is added in the range  $[v_k, v_p]$ . For each value, we access the corresponding entry in the binary relation corresponding to  $att_i$ . Then, we take the union of the tid-lists of these entries and  $L_i^j$ . The resulting number of tids due to this action is the number of tuples that satisfies to the generalized expression.

To determine the number of tuples satisfying to a specialization of  $e_i^j$ , we apply a similar procedure. We determine the values that is discarded in the range  $[v_k, v_p]$ . For each value, we access the corresponding entry in the binary relation corresponding to  $att_i$ . Finally, we take the difference of the tid-lists of these entries and  $L_i^j$ . The resulting number of tids due to this action is the number of tuples that satisfies to the specialized expression.  $\square$

**Cross-over** A crossover operation takes as input 2 expressions, it selects a random point and the subexpressions behind this point are exchanged. For example, a crossover on two expression  $e = \langle e_1^i \wedge e_2^i \wedge e_3^i \wedge \dots \wedge e_{k-1}^i \wedge e_k^i \wedge e_{k+1}^i \wedge \dots \wedge e_n^i \rangle$  and  $e' = \langle e_1^j \wedge e_2^j \wedge e_3^j \wedge \dots \wedge e_{k-1}^j \wedge e_k^j \wedge e_{k+1}^j \wedge \dots \wedge e_n^j \rangle$  at point  $k$  results into two new expressions, namely  $e'' = \langle e_1^i \wedge e_2^i \wedge e_3^i \wedge \dots \wedge e_{k-1}^i \wedge e_k^j \wedge e_{k+1}^j \wedge \dots \wedge e_n^j \rangle$  and  $e''' = \langle e_1^j \wedge e_2^j \wedge e_3^j \wedge \dots \wedge e_{k-1}^j \wedge e_k^i \wedge e_{k+1}^i \wedge \dots \wedge e_n^i \rangle$ .

In contrary to the operators discussed so far, no optimization guidelines can be given for this operator. The reason is that a cross-over operator randomly jumps from one state to another state in the search space.  $\square$

### 4.3 Algorithms

In Section 3, we have observed that the discussed search strategies choose from a current expression another expression for further exploration. They differ in the way the choice of the expression for further exploration is made. In this section, we consider the following three cases. In the first case, a hill climber and a simulated annealing algorithm are equipped with the extension operator, and in the second case both types of algorithms are equipped with the neighbour operator. In the third case, a genetic algorithm equipped with a mutation and cross-over operation is considered.

**Case 1a: hill climber** In Figure 3, we have depicted the search process of a hill climber. The different elementary expressions in which  $att_i$  is involved are distinguished by a superscript. So, the expression  $e_i^j$  represents the  $j$ -th expression in which  $att_i$  is involved. We assume that the generation of elementary expressions has been done in a separate process.

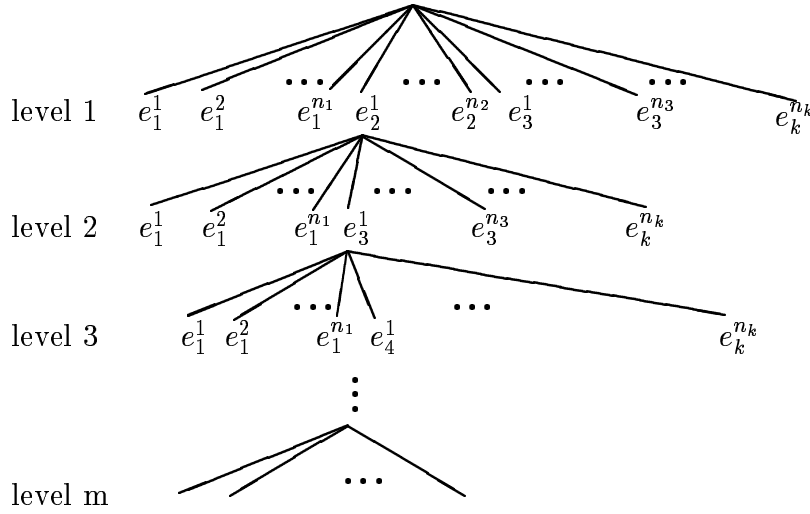


Figure 0.3: Search process of a hill climber

Techniques to generate these expressions can be found in [SrAg 96]. The hill climber starts with the evaluation of all elementary expressions  $e_i^j$  and chooses the expression with the highest quality. Then, it computes the extensions of the chosen expression; choose the extension with the highest quality, and the whole process is repeated again. So, the expressions that are considered in this process are conjunctions of operations along a path.

Since each non elementary expression considered in this search process at a level  $i$  is an extension of a certain expression  $e$  at level  $i - 1$ , we can use the optimization techniques for the extension operator as described in Section 4.2.

It should be clear that if we store for each elementary expression the tids of tuples that satisfy the expression and the tids of the tuples satisfying the expression whose extensions will be further explored, the main activity in processing queries is reduced to the intersection of 2 tid lists.

In order to store the list of tids that satisfies the expression that will be further explored, one should know this expression. Because this expression is not known at proper time, this may lead to the storage of many lists of tids. For example, while computing the extensions of the expression  $\langle e_2^1 \wedge e_3^1 \rangle$  in Figure 3, we do not know which of the extensions will be chosen for further exploration. Since we know that the hill climber will choose one of the extensions, we may decide to store for each expression at level 3 the list of tids satisfying the expression.

In general, as long as we do not know which extension of an expression  $e$  at level  $i - 1$  will be chosen for further exploration, one can decide to store all extensions of  $e$ . We note that the extensions of  $e$  are the expressions at level  $i$ . This means that if the number of branches at level  $i$  are  $b_i$ , we need extra storage space for  $b_i$  lists of tids. We note that the maximal number of branches is generated at level 1. Since we do not have to store tid lists satisfying elementary expressions, the maximal number of tid lists that should be stored is generated at level 2. We note that, in general, the longer the length of an expression will be, the shorter

the list of tids will be that satisfy the expression. This means that the longest lists will be also generated at level 2. Furthermore, once we know which expression is chosen for further exploration at level  $i$ , the list of tids concerning this level can be discarded.

Another alternative is not to store any of the lists of tids computed at level  $i$ , until we know which expression will be further explored. This information is released at the moment when the hill climber requires the evaluation of expressions at level  $i + 1$ . At that moment, it will ask to compute queries with regard to the extensions of the selected expression at level  $i$ . Since we have not stored any tid lists of tuples at level  $i$ , this means we have to compute the tid list of tuples that satisfies to the selected expression at level  $i$  again.

Consider the expression  $\langle e_2^1 \wedge e_3^1 \rangle$  at level 2 in Figure 3. Suppose that for the first time a query has an expression of length 4 in its WHERE clause. Let us assume that this expression is  $\langle e_2^1 \wedge e_3^1 \wedge e_1^{n_1} \wedge e_5^1 \rangle$ . Then, the selected expression at level 3 is  $\langle e_2^1 \wedge e_3^1 \wedge e_1^{n_1} \rangle$ . Since we have not stored any of the tid lists at level 3, we compute the following intersection again:  $L_{2,3,1} = L_{2,3} \cap L_1^{n_1}$ , in which  $L_{2,3} = L_2^1 \cap L_3^1$ , and store the list  $L_{2,3,1}$ . In this case, extra storage space is only required for *one* list of tids, namely those tids that satisfy the expression whose extensions will be further explored. On the other hand, this strategy requires one extra intersection between 2 tid lists at each level of the search process. Whenever the intersection of tid lists appears to be cheaper than temporary storing all the generated lists of tids at each level, the intersection of tid lists is preferred. Otherwise, the storage of tid lists is preferred.

We note that if we consider  $p, p > 1$ , expressions for further exploration, e.g., in case of a beam search, storage space is required for  $p$  lists of tids and  $p$  extra intersections of tid lists are required at each level. Depending on  $p$ , it may be sensible to store a limited number of the generated lists of tids at each level. In this way, the extra number of intersections at each level can be reduced.

**Case 1b: simulated annealing** A simulated annealing algorithm randomly choose an expression. Then, it selects an extension of this expression, and decides immediately whether this extension will be chosen for further exploration or not. If an extension is selected, this procedure is repeated. In Figure 4, a search process of a simulated annealing is depicted. Since this algorithm immediately determines whether an expression will be chosen or not, it is sufficient the store only one list of tids, namely the tid list of tuples satisfying the expression whose extensions are currently explored.

**Case 2:** In this case, we assume that a hill climber and a simulated annealing algorithm are equipped with the neighbour operation. As has been shown, the tid list of tuples satisfying an extension of an expression  $e$  can be computed by using the tid list of tuples satisfying  $e$ . In case of a reduction of  $e$ , the tid list of tuples satisfying  $e$  can not be used in computing the tid list of tuples satisfying the reduction.

We note that a neighbour operator in combination with a hill climber algorithm offers the possibility to leave an earlier chosen path and to explore a new one. Suppose that the application of the reduction operator to  $\langle e_2^1 \wedge e_3^1 \wedge e_1^{n_1} \rangle$  yields the expression  $\langle e_3^1 \wedge e_1^{n_1} \rangle$ . Then, in Figure 3, this concerns a path that starts at  $e_3^1$  at level 1.

In a simulated annealing algorithm as well as in a hill climber algorithm, it is possible that

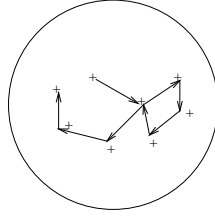


Figure 0.4: Search process of simulated annealing

an earlier visited expression will be visited again, or one of its extension will be visited. By storing lists of tids of expressions that are computed earlier in the search process, the number of lists of tids that should be intersected further on in the search process may be reduced. Consider the following expression  $e = \langle e_1 \wedge e_2 \wedge e_3 \wedge \dots \wedge e_j \wedge \dots \wedge e_n \rangle$ , and suppose that the reduction  $e' = \langle e_1 \wedge e_2 \wedge e_3 \wedge \dots \wedge e_{j-1} \wedge e_{j+1} \wedge \dots \wedge e_n \rangle$  is selected for further exploration. If  $e'$  was visited earlier and the corresponding tids satisfying  $e'$  has been stored, this can be reused in processing queries with  $e'$  in their WHERE clause. If, for example, not  $e'$  but  $e'' = \langle e_1 \wedge e_2 \wedge e_3 \wedge \dots \wedge e_{j-1} \rangle$  was visited earlier, then the list of tids satisfying  $e''$  can be used in computing the tid list satisfying  $e'$  ( $L_{e'}$ ), namely  $L_{e'} = L_{e''} \cap L_{j+1} \cap \dots \cap L_n$ .

It should be clear, the more of the computed tid lists during the search process are stored, the better the chances are that the number of lists that should be intersected can be reduced. However, since the available amount of storage space will be limited, it will be not possible to store all computed tid lists during the search process. A possible heuristic is to delete lists of tids of tuples that satisfy expressions that are not a subexpression of the expression that will be further explored. The rationale behind this heuristic is that if neighbours of an expression  $e$  become shorter, the tid lists of tuples that satisfy subexpressions of  $e$  can be used in computing queries having these neighbour expressions in their WHERE clause. If neighbours of  $e$  become longer, the tid lists of tuples that satisfy  $e$  can be used in computing queries having these neighbour expressions in their WHERE clause.

Another heuristic to discard tid lists if the available storage space is limited, is based on the length of tid lists. Tid lists with relatively few number of tids or a large number of tids in comparison with the cardinality of the database can be discarded. The rationale behind this heuristic is based on the fact that the quality of an expression is based on the number of tuples satisfying the expression. An expression to which only a few tuples satisfy will be in general not interesting, and, therefore, it will have a low quality. The same holds for expressions that yield almost the whole database.  $\square$ .

**Case 3:** A genetic algorithm starts with an initial population, i.e., a number of expressions in our terminology. It evaluates all expressions of the population and selects some of the expressions on which the mutation or the crossover operation is applied, yielding a new population. For each expression in a new generation holds; the expression is the same as in the previous generation or the expression is modified due to a crossover or a mutation.

It should be clear to store the list of tids satisfying to expressions of the present generation. A number of these expressions will remain the same in the next generation. So, we can reuse these tid lists in processing queries that have expressions in their WHERE clause which

remain the same in two consecutive generation.

The optimization techniques that have been discussed in Section 4.2 with regard to generalization and specialization can be applied whenever expressions undergo a mutation. To speed up queries which regard to expressions that are due to a cross-over, no general guidelines can be given. One can store tid lists that satisfy to subexpressions of an expression that appear in a present generation. Then, these tid lists may be used in the same way as in Case 2. The guidelines mentioned in Case 2 to control the storage space can be used here as well.  $\square$

We have analysed the role of a physical schema, the operators, and the algorithms in the optimization of queries generated by different search strategies. We have suggested to store a relation according to the binary storage model or to allocate a sorted index to each attribute. We have argued that the processing of queries may be accelerated by storing proper lists of tids. The amount of extra storage space required to store tid lists depends on the algorithm and operators used, e.g., hardly extra storage space is required for a simulated annealing algorithm that is equipped with an extension operator. In the case of a limited amount of storage space, we have introduced two heuristics to control the storage space.

In the next section, we discuss a framework of an optimization module in which above mentioned techniques are embedded.

## 5. FRAMEWORK

We present a framework of an optimization module for query optimization to support data mining. Furthermore, we show how the framework can be related to current database management systems.

The optimization module is depicted in Figure 5. The module receives as input a set of queries and the search strategy that generates the queries. A dispatcher passes the queries to an optimizing submodule depending on the search strategy. The optimizing submodule generates a piece of intermediate code for each query  $q$ . This module checks whether stored lists of tids can be used in computing the results of  $q$ , and determines which parts of the results should be stored. Furthermore, it also invokes heuristics to discard lists of tids whenever there is a shortage of storage space.

The intermediate code is passed to a translator, which translate the code in a language that is understood by the underlying database management system, e.g., SQL. Finally, the query is passed to the database management system. The query optimizer of the database management system may generate an efficient query execution plan. The result produced by the database management system is passed to the optimizing module, which passes it on its turn to the search strategy.

Let us illustrate the working of the optimization module by means of an example. Consider the earlier mentioned insurance database in Section 2. The database is stored according the binary storage model, and consists of, among others, the binary relations *age*, *gender*, and *accident*. Suppose that a simulated annealing search strategy passes the queries: count the tuples satisfying the expression  $e_1 = \langle \text{age} \in [19, 24] \wedge \text{gender} = \text{'female'} \rangle$  and count the tuples satisfying the expression  $e_2 = \langle \text{age} \in [19, 24] \wedge \text{gender} = \text{'female'} \wedge \text{accident} = \text{'true'} \rangle$ . Then, the dispatcher passes these queries to the submodule simulated annealing. This sub-

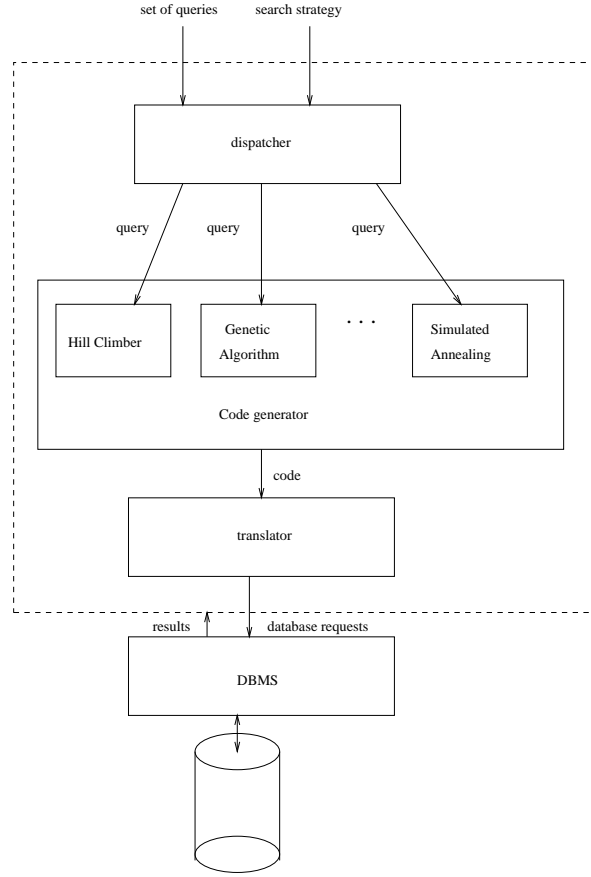


Figure 0.5: Diagram of the optimization framework

module checks whether it can accelerate the processing of these queries by earlier stored intermediate results, and generates a piece of code. Since no results are stored yet, the submodule simulated annealing generates the following intermediate code for the first query:

- (1)  $L_1 = \text{age.select}(19, 24);$
- (2)  $L_2 = \text{gender.select}('male');$
- (3)  $L_{e_1} = \text{intersect}(L_1, L_2);$
- (4)  $\text{pass}(\text{count}(L_{e_1}));$
- (5)  $\text{store}(L_{e_1});$

Since the list  $L_{e_1}$  has been stored and it can be used in processing the second query, the submodule simulated annealing generates the following code:

- (6)  $L_3 = \text{accident.select}('true');$
- (7)  $L_{e_2} = \text{intersect}(L_3, L_{e_1});$
- (8)  $\text{pass}(\text{count}(L_{e_2}));$
- (9)  $\text{store}(L_{e_2});$

We note that the statements (1), (2), and (6) can be done in parallel. Once  $L_3$  has been computed, statements (4) and (8) can be done in parallel too.

Depending on the underlying database management system, the translator translates the intermediate code in a language that is understood by the database management system. For example, if the database management system understand SQL, then the translator generates SQL queries. The SQL queries are offered to the database management system, which selects an efficient execution plan for them. In this way, we combine the optimizing techniques used by an optimizer and techniques based on reuse of information.

## 6. CONCLUSIONS & FURTHER RESEARCH

Many data mining problems can be characterized as the search for specific expressions among an enormous number of expressions, making an exhaustive search infeasible. The evaluation of each expression leads to a number of queries to the database to be mined. Although efficient search strategies attempt to minimize the number of queries to be evaluated, still many queries have to be evaluated to find the specified expression(s). Inefficient evaluation of these queries will have its impact on the performance of a whole data mining system, making such a system less valuable.

Since queries generated in a future step in a search process are dependent of queries generated at the present step, exploiting the dependencies between queries in a data mining session promises a considerable speed-up of the discovery process. In this paper, we have argued how such a speed-up can be achieved for the cost of some extra storage for five cases. Generalizing from these cases, we propose an optimization framework in which the “browsing optimization” seamlessly fits in the traditional query optimizing strategy.

A topic for the near future is the implementation of the framework and the connection of the module to commercial database systems as well as to experimental database management systems. Since optimizers of current database systems are not focussed on the optimization of aggregate queries, this will be another topic for future research.

## REFERENCES

- [AGII 92] Agrawal, R., Ghosh, S., Imielinski, T., Iyer, B., Swami, A., An Interval Classifier for Database Mining Applications, in Proc. of the 18th Very Large Data Base, 1992, pp. 560-573.
- [AgIS 93] Agrawal, R., Imielinski, T., Swami, A., Mining Association Rules between Sets of Items in Large Databases, in Proc. ACM SIGMOD '93 Int. Conf. on Management of Data, pp. 207-216.
- [AgSr 94a] Agrawal, R., Srikant, R., Fast Algorithms for Mining Association Rules, in Proc. Int. Conf. on Very Large Databases, pp 487-499.
- [AgSr 94b] Agrawal, R., Srikant, R., Mining Sequential Patterns, in Proc. 11th Int. Conf. on Data Engineering, 1995 pp. 3-14.
- [AuVK 95] Augier, S., Venturini, G., Kodratoff, Y., Learning First Order Logic Rules with a Genetic Algorithm, in Proc. 1st Int. Conf. on Knowledge Discovery and Data Mining, pp. 21-26.
- [Choe 95] Choenni, R., On the Automation of Physical Database Design, Ph.D. thesis, University of Twente, May 1995.

- [FPSU 96] Fayyad, U.M., Piatetsky-Shapiro, G., Smyth, P., Uthurusamy, R., (Eds), *Advances in Knowledge Discovery and Data Mining*, AAAI/The MIT Press, 1996.
- [GrSt 82] Grimmet, G.R., Stirzaker, D.R., *Probability and Random Processes*, Oxford Science Publications, Oxford University Press, New York, USA.
- [HaCC 92] Han, J., Cai, Y., Cerone, N., *Knowledge Discovery in Databases: An Attribute-Oriented Approach*, in *Proc. of the 18th Very Large Data Base*, 1992, pp. 547-559.
- [HoKe 94] Holsheimer, M., Kersten, M.L., *Architectural Support for Data Mining*, in *Proc. AAAI-94 Workshop on Knowledge Discovery*, pp. 217-228.
- [HoSw 95] Houtsma, M., Swami, A., *Set-Oriented Mining for Association Rules in Relational Databases*, in *Proc. 11th Int. Conf. on Data Engineering*, 1995 pp. 25-33.
- [MaTV 94] Mannila, H., Toivonen, H., Verkamo, A.I., *Efficient Algorithms for Discovering Association Rules*, in *Proc. AAI-94 Workshop on Knowledge Discovery*, pp.
- [Mich92] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, New York, USA.
- [NaSS 86] Nahar, S., Sahni, S., Shragowitz, E., *Simulated Annealing and Combinatorial Optimization*, in *Proc. 23rd Design Automation Conference*, pp. 293-299.
- [SWKH 76] Stonebraker, M., Wong, E., Kreps, P., Held, G., *The Design and Implementation of Ingres*, *ACM. Trans. on Database Systems* 1(3), pp. 189-222.
- [SrAg 96] Srikant, R., Agrawal, R., *Mining Quantitative Association Rules in Large Relational Tables*, to appear in *Proc. ACM SIGMOD '96 Int. Conf. on Management of Data*.
- [Ullm 89] Ullman, J.D., *Principles of Database and Knowledge-Base Systems, Vol.2: The New Technologies*, Computer Science Press, New York, USA, 1989.
- [WWVS 96] Wrobel, S., Wettschereck, D., Verkamo, I., Siebes, A., Mannila, H., Kwakkel, F., Kloesgen, W., *User Interactivity in Very Large Scale Data Mining*, to appear.